UNIVERSITAT POLITECNICA DE
CATALUNYA

PARALLELISM

Lab 2: Brief tutorial on OpenMP programming model

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

20th March 2019, Q1

Contents

1 Introduction

Talk about the Mandelbrot Set and about what we are going to do in this 3 sessions, in order.

2 Task decomposition analysis with *Tareador*

In this section, we had to analyse the two possible task granularities that could be exploited in the given program. To do it, we used the *Tareador* tool, which was very useful to see graphically the created tasks and which dependences are between them.

2.1 Point decomposition

In this decomposition strategy, a task corresponds with the computation of a single point (row, col) of the Mandelbrot set. Thus, we will have $row \times col$ tasks.

In order to analyse the potential parallelism of this strategy, we modified the code in *mandel-tar.c* to create several *Tareador* tasks. As we are using a point decomposition strategy, the tasks are created inside the most inner loop of the function mandelbrot. Figure 1 shows the fragment of the function that we modified.

```
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        tareador_start_task("point");

        ...

        tareador_end_task("point");
    }
}
```

Figure 1: Modified fragment of the *mandel-tar.c* code.

Once we did this, we executed interactively mandeld-tar and mandel-tar using run-tareador.sh script. The first one is used for timing purposes and to check for the numerical validity of the output (-o option) whereas with the second one we can visualize the Mandelbrot set.

The script has defined inside the size of the image to compute (-w option). In this case, the value was 8 to generate a reasonable task graph in a reasonable execution time. Hence, as we said before, the number of tasks will be $8 \times 8 = 64$.

Figures 2 and 3 below show the two task decomposition graphs (TDG) of the two possible executions. There are some nodes bigger than the others. This is because these nodes have executed more instructions than the rest. Therefore, these nodes represent pixels in white[1].

[1]The bigger the node, the more iterations it does in the do while fragment of the function. This make that the computed color is white.

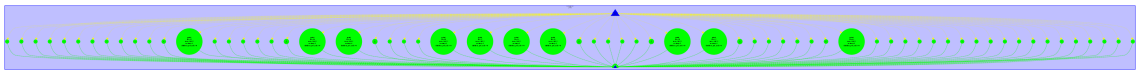

Figure 2: Mandel-tar task decomposition graph using the point decomposition strategy.

Figure 3: Mandeld-tar task decomposition graph using the point decomposition strategy.

This strategy will have more overhead of creation and termination of tasks than in the row strategy because it has to create more tasks. However, as a positive point the tasks are better distributed because if in a row there are many white areas, this work will not only be done by a single thread. Thus, it may end the execution earlier than in the row decomposition strategy.

2.2 Row decomposition

In this other strategy, a task corresponds with the computation of a whole row of the Mandelbrot set. This strategy only creates *row* tasks.

In this case, the code is not the same as before. We changed the creation of the *Tareador* tasks so that each time we enter a new row (second loop) a new task is created. The modified version of the code is shown below.

```
for (row = 0; row < height; ++row) {
    tareador_start_task("row");
    for (col = 0; col < width; ++col) {
        ...
    }
    tareador_end_task("row");
}
```

Figure 4: Modified fragment of the *mandel-tar.c* code.

Afterwards, we executed interactively mandel-tar and mandeld-tar again. We used the same size as before. Hence, the total number of tasks will be 8. The graphical results we got can be seen in figures 5 and 6.

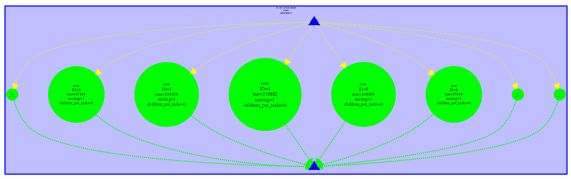

Figure 5: Mandel-tar task decomposition graph using the row decomposition strategy.

Figure 6: Mandeld-tar task decomposition graph using the row decomposition strategy.

In this strategy, the overhead time of creation and termination of tasks will be very small in comparison with the point strategy because it has to create less tasks. Nevertheless, it may happen that in a full row, all of its pixels must be white. In this case, it can happen that while other threads have finished their work, this other thread is still executing the row. Consequently, it is possible that the execution time may be bigger than in the point decomposition strategy.

2.3 Characteristics of the TDG

We saw in the previous sections that the execution of the mandel-tar has a very different task dependence graph than the execution of the mandeld-tar.

On the one hand, we can see that in mandel-tar every point is independent from the others. Consequently, we could parallelize that fragment of the code using OpenMP clauses.

On the other hand, in mandeld-tar we can see that all iterations have became sequential. In this situation, we do not gain anything by parallelizing the code, but we increase the execution time because of the overhead of creation and termination of tasks.

Using the *Tareador* we could see which variable was responsible of creating those dependences. We did the following: Right Click into a task -> Data View -> Edges-out -> Real Dependency. Figure 11 shows the result we got.

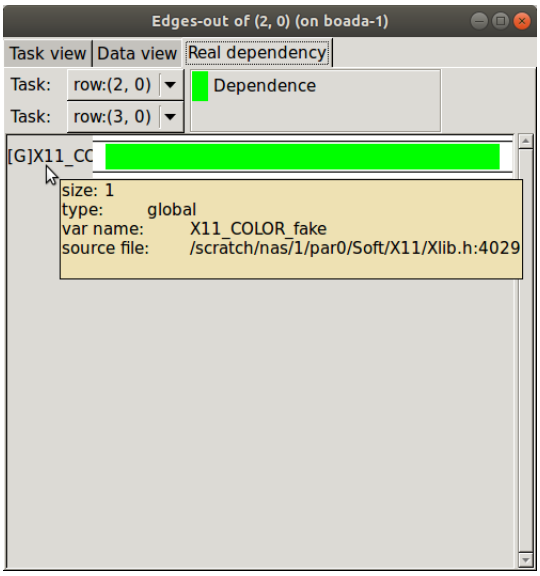

Figure 7: Variable that provoques the dependences between tasks.

We can see that there is only one variable that is causing all the dependences: X11_COLOR_fake. Observing the code, we noticed that the only difference between mandel-tar and mandeld-tar was a fragment of the code that was only executed in mode _DISPLAY_ (mandeld-tar):

```
#if _DISPLAY_
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;
    if (setup_return == EXIT_SUCCESS) {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
    }
#else
    output[row][col]=k;
#endif
```

Figure 8: Fragment of the *mandel-tar.c* code.

Therefore, variable X11_COLOR_fake is used at least in one of the functions XSetForeground and XDrawPoint. We could protect this section of code in the parallel OpenMP code using the *critical* clause to define a region of mutual exclusion where only one thread can be working at the same time.

```
#if _DISPLAY_
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;

    #pragma omp critical
    if (setup_return == EXIT_SUCCESS) {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
    }
#else
    output[row][col]=k;
#endif
```

Figure 9: Fragment of the *mandel-tar.c* code using the *critical* clause to protect a fragment of the code.

FALTA AIXO: Reason when each strategy/granularity should be used.

You also have to deliver the complete C source codes for Tareador instrumentation and all the OpenMP parallelization strategies that you have done. Include both the PDF and source codes in a single compressed tar file (GZ or ZIP). Only one file has to be submitted per group through the Raco website.

3 Point decomposition in OpenMP

In this section, we are going to explore different options in the OpenMP tasking model to express the Point decomposition for the Mandelbrot computation program. We will analyse the scalability and behaviour of these options.

3.1 OpenMP Task Implementation

The aim of this tasking model strategy is to create a task for each point. Figure 10 show the OpenMP clauses we used to implement the task strategy (the code can be found in file *mandel-omp-task-point.c* in codes folder).

```
for (int row = 0; row < height; ++row) {
    #pragma omp parallel
    #pragma omp single
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(col)
        {
            ...
            #pragma omp critical
            if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
                XDrawPoint (display, win, gc, col, row);
            }
            ...
        }
    }
}
```

Figure 10: Fragment of the *mandel-omp-task-point.c* code showing the OpenMP clauses to implement the task strategy.

Only one thread is creating all the tasks and inserting them into a pool of tasks, while the other threads are taking tasks from the pool and executing them. Thus, each iteration of the col loop will be executed as an independent task. The firstprivate clause is used in order to avoid problems of data races[2]. Finally, the critical clause is used to honour the dependences we detected for the graphical version in the previous section. However, this section of the code will not be called since we will execute the code without the graphical version (only mandel-tar).

The following figure shows the execution flow of the program using the *Paraver* tool.

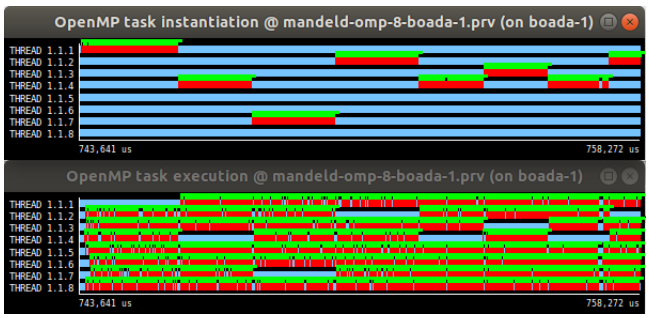

Figure 11: Zoomed part of the execution flow using the task strategy.

[2]We have not used the private clause because it initializes the variable with a random value, and we wanted the real value of the variable (iteration).

We can observe the effect of the single clause. In the upper part (task instantiation) only one thread is creating the tasks and inserting them in the task pool. In the bottom part (task execution) we can see that the other threads execute the tasks as they are put in the pool. The role of the task creator changes every time we finish a row.

In total, there are 640000 tasks created and executed. This is because the image has size 800×800. We have seen this using the "OMP_parallel_functions.cfg" i "OMP_state_profile.cfg" configurations. Figure XXXXXXXXXXXXXXX shows the results obtained.

The parallel construct is executed 1 time and the single worksharing construct is executed 1 time for each thread. All threads will execute the single lines but only one will "gain". This thread will be responsible for creating the tasks of that row.

Finally, figures 12 and 13 show the time and speedup plots.

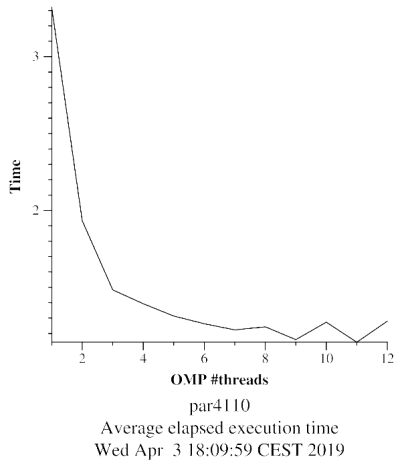

Figure 12: Execution time plot varying the number of threads.

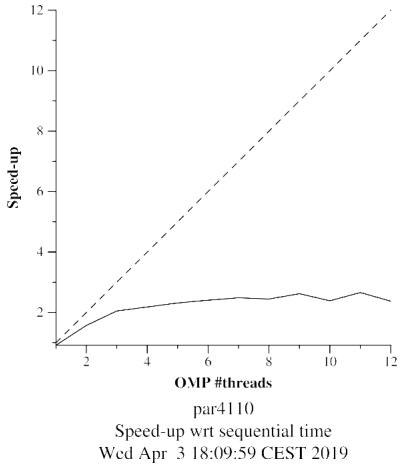

Figure 13: Speedup plot varying the number of threads.

In the first plot, the execution time is reduces as we increase the number of threads until the time stays constant. However, we think that if we increase a lot the number of threads, the execution time would end up increasing because of the overheads.

On the other hand, in the second plot the speedup increases a lot with 2-4 threads but ends up constant as we increase the number of threads.

REPASSAR EL SEGUENT: The scalability is not appropiate because the speedup stays at 2 (reduction of time of T_1) even though we have 8 threads.

3.1.1 OpenMP Taskwait Variant

In this variant of the previous code, only one thread (the one that gets access to the single region), traverses all iterations of the row and col loops, generating a task for each iteration of the innermost loop (point). To do it, we have introduced "#pragma omp taskwait" at the end of each iteration of a row. Consequently, the creator thread must wait until all tasks for a row finish. After that, the thread will

advace one iteration of the row loop and generate a new bunch of tasks. This new version of the code can be found in *mandel-omp-task-point-taskwait.c*.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
        {
            ...
        }
    }
    #pragma omp taskwait // waiting point for all child tasks
}
```

Figure 14: Fragment of the *mandel-omp-task-point-taskwait.c* code showing the OpenMP clauses to implement the task strategy with the taskwait variant.

The number of created and executed tasks is still 640000 as we have not modified the size of the image. Nevertheless, even though the number of calls to parallel is the same, the number of calls to the single worksharing construct will only be 8, the number of threads. Besides, the taskwait clause will be executed 800 times (number of rows). The granularity is the same than before. Each task has exactly one iteration of the for.

Figures 15 and 16 show again the time and speedup plots. We have not noticed any significant difference between these plots and the plots of the previous section.

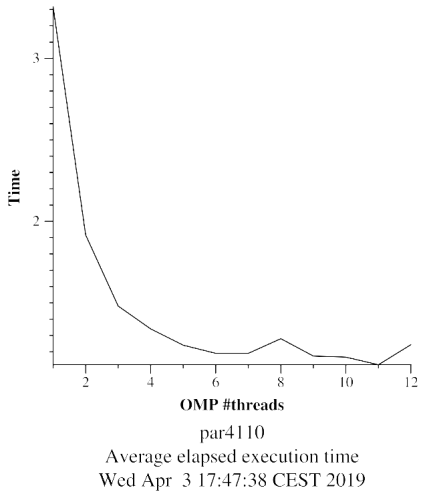

Figure 15: Execution time plot varying the number of threads.

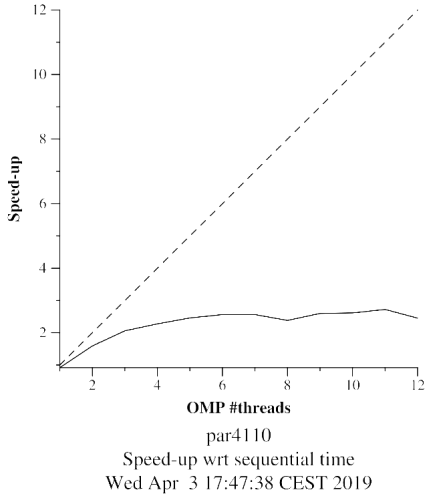

Figure 16: Speedup plot varying the number of threads.

3.1.2 OpenMP Taskgroup Variant

In this other variant, we define a region in the program where the thread will wait for the termination of all descendant (not only child) tasks. Figure 17 shows a section

of the modified code that can be found in file *mandel-omp-task-point-taskgroup.c* inside the codes directory. Now, the pixels are generated without any specific order.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskgroup
    {
        for (int col = 0; col < width; ++col) {
            #pragma omp task firstprivate(row, col)
            {
                ...
            }
        }
    }
}
```

Figure 17: Fragment of the *mandel-omp-task-point-taskgroup.c* code showing the OpenMP clauses to implement the task strategy with the taskgroup variant.

Figures 18 and 19 show again the time and speedup plots. We have not noticed any significant difference between these plots and the plots of the two previous sections.

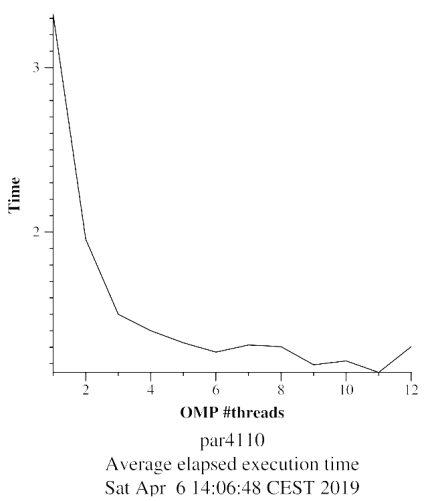

Figure 18: Execution time plot varying the number of threads.

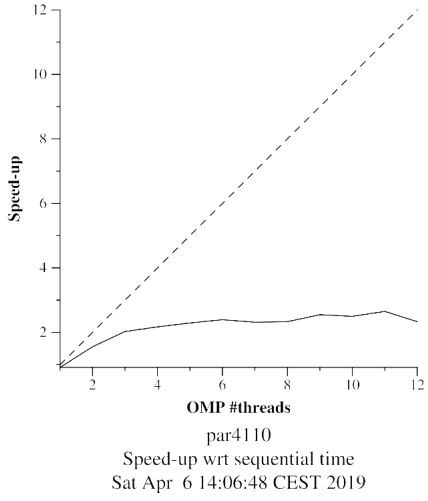

Figure 19: Speedup plot varying the number of threads.

FALTA: Do you think the taskwait construct in Figure 2.2 is really necessary? Or in other words, is it necessary to wait for the termination of all tasks in a row before generating the tasks for the next rows? Modify the code in mandel-omp.c to remove this task barrier and repeat the previous evaluation (scalability and tracing). Has the number of tasks created/executed changed? Why the threads generating tasks

stops task generation, then executes some tasks, and then proceeds generating new tasks?

El taskwait no es necessari ja que les taskes no tenen res a veure unes amb les altres a la hora de calcular. Per tant tan es l'ordre de generacio de les tasques, entre diferents rows. Em observat que ens dona els mateixos resultats amb l'execució anterior. El nombre de taskes creades i executades no varien respecte l'execució anterior. Això es culpa de que la bossa de taskes que té l'omp és limitada, per això arriba un punt que de forma intel·ligent el thread 0 atura la creació de taskes, per executar-ne i així poder disminuir la quantitat total de taskes a la bossa.